

Java 1.5 & Effective Java



Fawzi Emad
Chau-Wen Tseng

Department of Computer Science
University of Maryland, College Park

Java 1.5 (Tiger)

■ Description

- Released September 2004
- Largest revision to Java so far

■ Goals

- Less code complexity
- Better readability
- More compile-time type safety
- Some new functionality (generics, scanner)

New Features in Java 1.5

- Generic types
- Scanner
- Autoboxing & unboxing
- Enhanced for loop
- Variable number of arguments (varargs)
- Enumerated types
- Static imports
- Annotations

Generics – Motivating Example

- Problem
 - Utility classes handle arguments as Objects
 - Objects must be cast back to actual class
 - Casting can only be checked at runtime
- Example

```
class A { ... }
class B { ... }
List myL = new List();
myL.add(new A()); // Add an object of type A
...
B b = (B) myL.get(0); // throws runtime exception
// java.lang.ClassCastException
```

Generic Types

- **Parameterized types with <type parameter>**
 - Parameterize classes, interfaces, methods by types
 - Parameters defined using <x> notation
 - Parameters replaced at compile time with casts
 - Provide compile-time type safety
- **Support in java.util**
- **Example**
 - `public class foo<x, y, z> { ... }`
 - `public class List<String> { ... }`

Generics – Usage

- **Using generic types**
 - Specify <type parameter> for utility class
 - Automatically performs casts
 - Can check class at compile time
- **Example**

```
class A { ... }  
class B { ... }  
List<A> myL = new List<A>();  
myL.add(new A()); // Add an object of type A  
A a = myL.get(0); // myL element ⇒ class A  
...  
B b = (B) myL.get(0); // causes compile time error
```

Scanner

■ Iterator for

- Provides methods for input & parsing
- Supports String `nextLine()`, int `nextInt()`...
- Throws `InputMismatchException` if wrong format

■ Example

// old approach to scanning input

```
BufferedReader br = new BufferedReader( new  
    InputStreamReader(System.in));  
String name = br.readLine();
```

// new approach using scanner

```
Scanner in = new Scanner(System.in);  
String name = in.nextLine();
```

Autoboxing & Unboxing

■ Automatically convert primitive data types

- Data value \Leftrightarrow Object (of matching class)
- Data types & classes converted
 - Boolean, Byte, Double, Short, Integer, Long, Float

■ Example

```
ArrayList myL = new ArrayList();  
myL.add(1);    // previously myL.add(new Integer(1));
```

```
Integer X = new Integer(2);  
int y = X;    // previously int y = X.intValue();
```

Enhanced For Loop

- For loop handles Iterator automatically

- Test hasNext(), then get & cast next()

- Example

```
Iterator it = myL.iterator(); // old usage of Iterator
while (it.hasNext()) {
    Integer num = (Integer) it.next();
    // do something with num...
}
for (Integer num : myL) { // new enhanced for loop
    // do something with num...
}
```

Variable # of Arguments (Varargs)

- Method allow variable # of arguments (vararg)

- Arguments automatically stored in array
- Only single vararg allowed, must be last argument

- Example

```
void foo(int x, String ... myL) {
    for (String str : myL) {
        // do something with str...
    }
}
foo( 1, "car", "boat");
foo( 2, "car", "boat", "plane");
foo( 3, String [ ] x );
```

Enumerated Types

- **New type of variable with set of fixed values**
 - Establishes all possible values by listing them
 - Supports values(), valueOf(), name(), compareTo()...

- **Example**

```
public Class Color { // old approach to enumeration
    private int c;
    public static final Color Black = new Color(1);
    public static final Color White = new Color(2);
}
public enum Color { Black, White } // new enumeration
Color myC = Color.Black;
for (Color c : Color.values()) System.out.println(c);
```

Static Import

- **Import static members of package**

- **Example**

```
// imports static members of package
import static java.lang.Math.ceil

// imports all static members of package
import static java.lang.Math.*

double x, y;
x = ceil(y); // can use method name directly
```

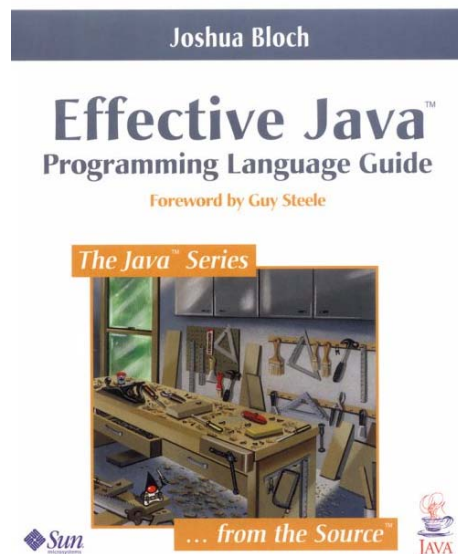
Annotations

- Add annotations (metadata) using @
 - Annotate types, methods, fields for documentation, code generation, runtime services
 - Provides built-in & custom annotations
 - @Target, @Overrides, @Documented...
 - Can control availability of annotations
 - Source code, class file, runtime in JVM
- Example

```
/* @author CMSC132Coder */
public final class AnnotationsTest {
    @Overrides
    public String toString(int i) { return " x "; }
}
```

Effective Java

- Title
 - Effective Java Programming Language Guide
- Author
 - Joshua Bloch
- Contents
 - Useful tips for Java programming



Effective Java – Topics

1. **Creating and Destroying Objects**
2. **Methods Common to All Objects**
3. **Classes and Interfaces**
4. **Substitutes for C Constructs**
5. **Methods**
6. **General Programming**
7. **Exceptions**
8. **Threads**
9. **Serialization**

Creating and Destroying Objects

- **Consider providing static factory methods instead of constructors**
- **Enforce singleton property with a private constructor**
- **Enforce noninstantiability with a private constructor**
- **Avoid creating duplicate objects**
- **Eliminate obsolete object references**
- **Avoid finalizers**

Methods Common to All Objects

- Obey the general contract when overriding equals
- Always override hashCode when you override equals
- Always override toString
- Override clone judiciously
- Consider implementing Comparable

Classes and Interfaces

- Minimize the accessibility of classes and members
- Favor immutability
- Favor composition over inheritance
- Design and document for inheritance or else prohibit it
- Prefer interfaces to abstract classes
- Use interfaces only to define types
- Favor static member classes over nonstatic

Methods

- Check parameters for validity
- Make defensive copies when needed
- Design method signatures carefully
- Use overloading judiciously
- Return zero-length arrays, not nulls
- Write doc comments for all exposed API elements

General Programming

- Minimize the scope of local variables
- Know and use the libraries
- Avoid float and double if exact answers are required
- Avoid strings where other types are more appropriate
- Beware the performance of string concatenation
- Refer to objects by their interfaces
- Prefer interfaces to reflection
- Use native methods judiciously
- Optimize judiciously
- Adhere to generally accepted naming conventions

Exceptions

- Use exceptions only for exceptional conditions
- Use checked exceptions for recoverable conditions and run-time exceptions for programming errors
- Avoid unnecessary use of checked exceptions
- Favor the use of standard exceptions
- Throw exceptions appropriate to the abstraction
- Document all exceptions thrown by each method
- Include failure-capture information in detail messages
- Strive for failure atomicity
- Don't ignore exceptions

Threads

- Synchronize access to shared mutable data
- Avoid excessive synchronization
- Never invoke wait outside a loop
- Don't depend on the thread scheduler
- Document thread safety
- Avoid thread groups